The Lambda Expression

The syntax of this lambda expression is on the left below.

This was passed directly as a method argument, for a parameter type that was a Comparator.

The Comparator's abstract method, compare, is shown here on the right side.

The generated Lambda Expression	Comparator's Abstract Method
`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`	`int compare(T o1, T o2)`

The lambda expression parameters are determined by the associated interface's method, the functional method.

In the case of a Comparator, and it's compare method, there are two arguments.

This is why we get o1, and o2 in parentheses, in the generated lambda expression.

These arguments can be used in the expression, which is on the right of the arrow token.

The Syntax of a Lambda Expression

A lambda expression consists of a formal parameter list, usually but not always declared in parentheses; the arrow token; and either an expression or a code block after the arrow token.

Because lambda expressions are usually simple expressions, it's more common to see them written as shown on this slide.

```
(parameter1, parameter2, ... ) -> expression;
```

The expression should return a value, if the associated interface's method returns a value.

In the case of our generated expression, it returns an int, which is the result of the compare method on Comparator.

```
(o1, o2) -> o1.lastName().compareTo(o2.lastName())
```

Comparing the anonymous class and the lambda expression

Are you asking, where's the link between the compare method, and this lambda expression?

It's obvious in the anonymous class, because we override the compare method, and return the result of that expression.

Anonymous Class	Lambda Expression

`new Comparator<Person>() {`
 	
`    @Override`
`    public int compare(Person o1, Person o2) {`
`        return o1.lastName().compareTo(o2.lastName());`
`    }`
`};`
	`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`

Comparing the anonymous class and the lambda expression

We can see the two parameters and their types, and what the return value should be, in the anonymous class.

But the lambda expression has no reference to an enclosing method, as far as we can see from this code.

Anonymous Class	Lambda Expression

`new Comparator<Person>() {`
 	
`    @Override`
`    public int compare(Person o1, Person o2) {`
`        return o1.lastName().compareTo(o2.lastName());`
`    }`
`};`
	`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`

Where's the method in the lambda expression?

For a lambda expression, **the method is inferred** by Java!

How can Java infer the method?

Java takes its clue from the reference type, in the context of the lambda expression usage.

Here, I show a simplified view, of the sort method on List.

```
void sort(Comparator c)
```

And here is the call to that method passing the lambda expression.

```
people.sort((o1, o2) -> o1.lastName().compareTo(o2.lastName()));
```

From this, Java can infer that this lambda expression, resolves to a Comparator type, because of the method declaration.

This means the lambda expression passed, should represent code for a specific method on the Comparator interface.

How can Java infer the method?

But which method?

Well, there's only one the lambda expression cares about, and that's **the abstract method** on Comparator.

Java requires types which support lambda expressions, to be something called a functional interface.

What's a functional interface?

A functional interface is an interface that has **one, and only one, abstract method**.

This is how Java can infer the method, to derive the parameters and return type, for the lambda expression.

You may also see this referred to as SAM, which is short for **Single Abstract Method**, which is called the functional method.

A functional interface is the **target type for a lambda expression**.